

```
--  
-----  
-- Byte controller section  
-----  
  
library IEEE;  
use ieee.std_logic_1164.all;  
use ieee.std_logic_unsigned.all;  
use ieee.std_logic_arith.all;  
library synplify;  
use synplify.attributes.all;  
  
entity i2c_master_byte_ctrl is  
    generic(  
        Tcq : time := 1 ns  
    );  
    port (  
        clk      : in std_logic;  
        nReset  : in std_logic; -- asynchronous active low reset (FPGA compatible)  
--        ena      : in std_logic; -- core enable signal  
--  
--        clk_cnt : in unsigned(15 downto 0); -- 4x SCL  
--  
--        input signals  
        start,  
        stop,  
        read,  
        write,  
        ack_in  : std_logic;  
        din     : in std_logic_vector(7 downto 0);  
--  
--        output signals  
        cmd_ack  : out std_logic;  
        ack_out   : out std_logic;  
        i2c_busy  : out std_logic;  
        dout      : out std_logic_vector(7 downto 0);  
--  
--        i2c lines  
--        scl_i     : in std_logic; -- i2c clock line input  
--        scl_o     : out std_logic; -- i2c clock line output  
--        scl_oen   : out std_logic; -- i2c clock line output enable, active low  
--        sda_i     : in std_logic; -- i2c data line input  
--        sda_o     : out std_logic; -- i2c data line output  
--        sda_oen   : out std_logic -- i2c data line output enable, active low  
    );  
end i2c_master_byte_ctrl;  
  
architecture behave_rtl of i2c_master_byte_ctrl is  
attribute syn_radhardlevel of behave_rtl : architecture is "tmr";  
  
component i2c_master_bit_ctrl is  
    --generic(  
    --    Tcq : time := Tcq  
    --);  
    port (  
        clk      : in std_logic;  
        nReset  : in std_logic;  
        ena      : in std_logic; -- core enable signal  
--  
        clk_cnt : in unsigned(6 downto 0); -- clock prescale value  
        cmd     : in std_logic_vector(3 downto 0);  
        cmd_ack : out std_logic;  
        busy    : out std_logic;  
--  
        din    : in std_logic;
```

```

dout : out std_logic;

-- i2c lines
scl_i  : in std_logic; -- i2c clock line input
scl_o  : out std_logic; -- i2c clock line output
scl_oen : out std_logic; -- i2c clock line output enable, active low
sda_i  : in std_logic; -- i2c data line input
sda_o  : out std_logic; -- i2c data line output
sda_oen : out std_logic -- i2c data line output enable, active low
);
end component i2c_master_bit_ctrl;

-- commands for bit_controller block
constant I2C_CMD_NOP    : std_logic_vector(3 downto 0) := "0000";
constant I2C_CMD_START   : std_logic_vector(3 downto 0) := "0001";
constant I2C_CMD_STOP    : std_logic_vector(3 downto 0) := "0010";
constant I2C_CMD_READ    : std_logic_vector(3 downto 0) := "0100";
constant I2C_CMD_WRITE   : std_logic_vector(3 downto 0) := "1000";

constant ena      : std_logic := '1';
constant scl_i    : std_logic := '1';
-- signals for bit_controller
signal core_cmd : std_logic_vector(3 downto 0);
signal core_ack, core_txd, core_rxd : std_logic;

signal clk_cnt : unsigned(6 downto 0);

-- signals for shift register
signal sr : std_logic_vector(7 downto 0); -- 8bit shift register
signal shift, ld : std_logic;

-- signals for state machine
signal go, host_ack : std_logic;
signal dcnt : unsigned(2 downto 0); -- data counter
signal cnt_done : std_logic;

begin
  -- hookup bit_controller
  u1: i2c_master_bit_ctrl port map(
    clk      => clk,
    nReset   => nReset,
    ena      => ena,
    clk_cnt  => clk_cnt,
    cmd      => core_cmd,
    cmd_ack  => core_ack,
    busy     => i2c_busy,
    din      => core_txd,
    dout     => core_rxd,
    scl_i    => scl_i,
    scl_o    => scl_o,
    scl_oen  => scl_oen,
    sda_i    => sda_i,
    sda_o    => sda_o,
    sda_oen  => sda_oen
  );
  clk_cnt <= "1000000"; -- 0064h or 100dec -- use 10dec for 1Mhz simulations
  -- generate host-command-acknowledge
  cmd_ack <= host_ack;

  -- generate go-signal
  go <= (read or write or stop) and not host_ack;

  -- assign Dout output to shift-register
  dout <= sr;

  -- generate shift register
  shift_register: process(clk, nReset)

```

```

begin
    if (nReset = '0') then
        sr <= (others => '0') ; -- after Tcq;
    elsif (clk'event and clk = '1') then
        if (ld = '1') then
            sr <= din ; -- after Tcq;
        elsif (shift = '1') then
            sr <= (sr(6 downto 0) & core_rxd) ; -- after Tcq;
        end if;
    end if;
end process shift_register;

-- generate data-counter
data_cnt: process(clk, nReset)
begin
    if (nReset = '0') then
        dcnt <= (others => '0') ; -- after Tcq;
    elsif (clk'event and clk = '1') then
        if (ld = '1') then
            dcnt <= (others => '1') ; -- after Tcq; -- load counter with 7
        elsif (shift = '1') then
            dcnt <= dcnt - 1 ; -- after Tcq;
        end if;
    end if;
end process data_cnt;

cnt_done <= '1' when (dcnt = 0) else '0';

--
-- state machine
--
statemachine : block
    type states is (st_idle, st_start, st_read, st_write, st_ack, st_stop);
    signal c_state : states;
begin
    --
    -- command interpreter, translate complex commands into simpler I2C commands
    --
    nxt_state_decoder: process(clk, nReset)
    begin
        if (nReset = '0') then
            core_cmd <= I2C_CMD_NOP ; -- after Tcq;
            core_txd <= '0' ; -- after Tcq;

            shift     <= '0' ; -- after Tcq;
            ld       <= '0' ; -- after Tcq;

            host_ack <= '0' ; -- after Tcq;
            c_state  <= st_idle ; -- after Tcq;

            ack_out   <= '0' ; -- after Tcq;
        elsif (clk'event and clk = '1') then
            -- initially reset all signal
            core_txd <= sr(7) ; -- after Tcq;

            shift     <= '0' ; -- after Tcq;
            ld       <= '0' ; -- after Tcq;

            host_ack <= '0' ; -- after Tcq;

            case c_state is
                when st_idle =>
                    if (go = '1') then -- go = read or write or stop
                        if (start = '1') then
                            c_state  <= st_start ; -- after Tcq;
                            core_cmd <= I2C_CMD_START ; -- after Tcq;
                        elsif (read = '1') then
                            c_state  <= st_read ; -- after Tcq;
                        end if;
                    end if;
                end case;
            end if;
        end if;
    end process;
end block statemachine;

```

```

        core_cmd <= I2C_CMD_READ ; -- after Tcq;
elsif (write = '1') then
    c_state <= st_write ; -- after Tcq;
    core_cmd <= I2C_CMD_WRITE ; -- after Tcq;
else -- stop
    c_state <= st_stop ; -- after Tcq;
    core_cmd <= I2C_CMD_STOP ; -- after Tcq;

        host_ack <= '1' ; -- after Tcq; -- generate acknowledg
e signal
end if;

        ld <= '1' ; -- after Tcq;
end if;

when st_start =>
    if (core_ack = '1') then
        if (read = '1') then
            c_state <= st_read ; -- after Tcq;
            core_cmd <= I2C_CMD_READ ; -- after Tcq;
        else
            c_state <= st_write ; -- after Tcq;
            core_cmd <= I2C_CMD_WRITE ; -- after Tcq;
        end if;

        ld <= '1' ; -- after Tcq;      -- load data to transmit
end if;

when st_write =>
    if (core_ack = '1') then
        if (cnt_done = '1') then
            c_state <= st_ack ; -- after Tcq;
            core_cmd <= I2C_CMD_READ ; -- after Tcq;
        else
            c_state <= st_write ; -- after Tcq;          -- stay in
same state
            core_cmd <= I2C_CMD_WRITE ; -- after Tcq; -- write ne
xt bit

        shift     <= '1' ; -- after Tcq;
    end if;
end if;

when st_read =>
    if (core_ack = '1') then
        if (cnt_done = '1') then
            c_state <= st_ack ; -- after Tcq;
            core_cmd <= I2C_CMD_WRITE ; -- after Tcq;
        else
            c_state <= st_read ; -- after Tcq;          -- stay in sa
me state
            core_cmd <= I2C_CMD_READ ; -- after Tcq; -- read next
bit
    end if;

        shift     <= '1' ; -- after Tcq;
        core_txd <= ack_in ; -- after Tcq;
    end if;

when st_ack =>
    if (core_ack = '1') then
        -- check for stop; Should a STOP command be generated ?
        if (stop = '1') then
            c_state <= st_stop ; -- after Tcq;
            core_cmd <= I2C_CMD_STOP ; -- after Tcq;
        else
            c_state <= st_idle ; -- after Tcq;
            core_cmd <= I2C_CMD_NOP ; -- after Tcq;

```

```
        end if;

        -- assign ack_out output to core_rxd (contains last received bit)
        ack_out  <= core_rxd ; -- after Tcq;

        -- generate command acknowledge signal
        host_ack <= '1' ; -- after Tcq;

        core_txd <= '1' ; -- after Tcq;
    else
        core_txd <= ack_in ; -- after Tcq;
    end if;

when st_stop =>
    if (core_ack = '1') then
        c_state  <= st_idle ; -- after Tcq;
        core_cmd <= I2C_CMD_NOP ; -- after Tcq;
    end if;

when others => -- illegal states
    c_state  <= st_idle ; -- after Tcq;
    core_cmd <= I2C_CMD_NOP ; -- after Tcq;
    report ("Byte controller entered illegal state.");
end case;

end if;
end process nxt_state_decoder;

end block statemachine;
end behave_rtl;
```